Radio Shack®

PRELIMINARY

INSTRUCTION MANUAL

DISK BASIC VERSION 1.1

TRSDOS VERSION 2.0

JULY 7 1978

This Manual describes the first release of TRSDOS. New releases will contain additions and new features not implemented in this preliminary version.

LIMITED WARRANTY

Radio Shack warrants for a period of 90 days from the date of delivery to customer that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if the computer case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair. Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

©Copyright 1978, by Radio Shack, A Division of Tandy Corporation, Ft. Worth, Texas 76102

THINGS YOU SHOULD KNOW

1. TRSDOS and DISK BASIC require 10K of RAM collectively.
2. After an INPUT# is performed from cassette, subsequent READ statements will automatically RESTORE data each time a READ is performed. To fix this, simply perform the statement POKE16553,255 before the first READ is performed. This pertains to LEVEL II only, not to DISK BASIC.
3. When perfroming an INPUT# from cassette, the maximum number of byte which can be read is 248. This does not affect disk operations.
4. If the RESET button is pressed when the expansion interface is attached to the TRS-80, any programs in memory will be lost.
5. If a BASIC program is stopped during execution, and alterations are made to the program, or EDIT mode is entered, then ALL VARIABLES will be set to zero. The program must be RUN again from the beginning.
6. If an LPRINT or LLIST is performed without a TRS-80 lineprinter being attached, the computer will "freeze-up". The user must press RESET or attach a lineprinter and turn it on.
7. All functions in LEVEL II BASIC always return single precision value (6-7 digits of accuracy). All trigonometric functions use or return radian angles. Use of degrees angles are described in the LEVEL II BASIC MANUAL.
8. All machine language programs currently available through Radio Shack will not function properly when used with DISK BASIC.
9. Frequent occurrences of SYNTAX errors may be caused by one of two subtle errors.
   a) If a letter or the at-symbol (@) were typed with the SHIFT key depressed, the letters will appear to be correct on the screen, but are really invalid. Try retyping the line, and beware of the SHIFT key.

b) Sometimes a space is required in a BASIC statement. All the following lines are incorrect

```
IFD < 0D=0
FIELD#1,20ASC$
```

The characters "0D" represent a double precision zero. "ASC" is a BASIC reserved word. The correct statements read (note the space and THEN)

```
IFD < 0 THEN D=0
FIELD#1,20AS C$
```

10. The format of a CLOAD? command to verify from cassette #2 is

```
CLOAD#-2,?"filename"
```

11. If the pressing of a key frequently causes multiple letters to be typed, the plastic key should be removed and the contacts beneath cleaned. Replace the plastic key when finished.

12. TRS-80 owner may phone Radio Shack Computer Services for answers to questions.

(817) 390-3583

or letters may be written to:

ATTN: HUGH MATTHIAS
Radio Shack Computer Services
P.O. Box 185
Fort Worth, TX 76102

13. When an INPUT from keyboard is executed, the line below the current cursor position is erased.

14. The following table summarizes the effect of opening a file (see p. 21):

mode	if file exists	if non-existent file
"I"	OK	FILE NOT FOUND error message
"O"	file will be over-written	file will be created
"R"	OK	file will be created

15. Disk file-space acquisition and release (SAVE and KILL) may not always work when the diskette is within 5K of being full. Use of the last 5K of free space on each diskette should be avoided in release 2.0 of TRSDOS.

LEVEL II DISK BASIC and TRSDOS

GENERAL INTRODUCTION

With the addition of your TRS-80 Disk Operating System (called TRSDOS), you now have three distinct yet related modes on your micro computer.

1) LEVEL II BASIC

   This is the same LEVEL II BASIC as described in the LEVEL II BASIC Reference Manual and is still available to you.

2) DISK BASIC

   With the information on the TRSDOS system floppy disk, your normal LEVEL II BASIC is extended into DISK BASIC, which can read/write data files and load/save programs to disk.

3) TRSDOS

   The Disk Operating System oversees operation of your disk drives, and provides powerful utilities such as copying one diskette to another, or listing all programs stored on a diskette.

The use of mini-disks will greatly expand the versatility of the TRS-80. Disks provide a fast and efficient method of accessing programs that would otherwise be stored on tape. They also provide a convenient method of storing data.

Data is accessed by DISK BASIC by either RANDOM or SEQUENTIAL methods. Sequential access methods are very similar to LEVEL II BASIC statements for storing data onto tape. RANDOM access may take a little longer to master but, the versatility and control it allows will make its use well worth the extra time devoted to learning it.

POWER UP AND OPERATING MODES

The Disk Operating System and DISK BASIC are stored on the System Diskette. This diskette is labelled TRSDOS and MUST always be in drive 0 (the drive closest to the expansion interface). Not all of DISK BASIC or TRSDOS are needed in RAM memory at the same time, therefore, the parts that are needed are copied to RAM when they are called for. This is why the TRSDOS diskette must be in drive zero at all times.

Turn on the power to the expansion interface and the disk drives. Gently insert the TRSDOS diskette into drive 0. Now press the power button on the back of your TRS-80 keyboard. The Disk Operating System will automatically load to RAM from drive 0. When this sequence is complete, the computer will respond:

```
TRSDOS-DISK OPERATING SYSTEM-VER 2.0
DOS READY
```

This is the command level of the Disk Operating System (DOS). Under this level, you are actually in a "system" mode and can perform the functions described in section 3 of this manual. Pressing the RESET button at anytime will return you to this point.

To use LEVEL II BASIC (without DISK BASIC extension), simply type:

```
BASIC2 and (ENTER)
```

The computer will respond with:

```
MEMORY SIZE?
```

LEVEL II BASIC will now operate as described in the LEVEL II BASIC Reference Manual. Pressing RESET will return you to TRSDOS, and be sure to save any program before doing so or they will be lost. However, you will probably want to use RADIO SHACK DISK BASIC. The command BASIC, when typed in after the DOS READY command will load DISK BASIC

into memory.

At this time you must specify the maximum number of files that will be open (in use) at the same time. DISK BASIC asks:

HOW MANY FILES?

Respond with the maximum number of files you wish to use. You may not use any more than 15 files at once. Each file you ask for sets aside a 256 byte buffer (more on buffers in sequential/random file usage). So a request of 4 files will reserve about 1K of memory. You can specify a default value of 3 files by pressing (ENTER).

The next question is:

MEMORY SIZE?

Respond with the highest address (in decimal only) availiable to DISK BASIC or press ENTER and DISK BASIC takes all memory it can find.

If at any time you wish to return to the Disk Operating System from DISK BASIC, type:

CMD"S"

The computer will respond with DOS READY. If you are in DISK BASIC, be sure to save any programs on tape or disk, BEFORE you return to TRSDOS or your program will be lost.

DISK DRIVES

The TRS-80 allows up to 4 mini-disk drives. The drives are numbered 0 to 3. Drive 0 is located on the cable closest to the expansion interface. If other disk drives are used, they are numbered sequentially (up to 3) as they occur on the cable. Disks will be referred to by their drive numbers in this manual.

DISKETTE CARE AND HANDLING

Diskettes are simply sheets of magnetic recording material specially prepared for use in the computer system. Diskette are particularly vulnerable to abuse and great care should be exercised when handling them. When not in the drive itself, the disk should be placed in its protective sheath. The user should avoid touching the recording surface exposed by the oval window in the paper cover. As in all recording material, the diskettes should be protected from dust, high temperatures and magnetic fields.

You may physically prevent a disk from being written on by "write protecting" it. This done by placing a small piece of tape (write tab) over the square notch on the mini-floppy. This will prevent any future recording on the mini-disk until the tape is removed from the notch.

Only one side of the TRS-80 mini-disk is used to record information. When placed in the drive, the square notch should be on the upper edge and the label opposite the red indicator light.

PHYSICAL DISKETTE FORMAT

TRSDOS mini-disks are formatted with 35 concentric circles where data is recorded. Each circle is called a "track". Each track is evenly divided into 10 sections called "sectors". Each sector contains room for recording 256 bytes of information. So each track contains 2560 bytes and each mini-disk contains 89600 bytes. TRSDOS can copy information from/to memory at a rate of 12.5K bytes per second. Not all 89K of the mini-disk are for your use. TRSDOS keeps a directory on each diskette so that the sectors related to a program or data file are known. This leaves the user about 85K of free space. The TRSDOS system diskette has 55K of free space on it.

MEMORY SIZE

TRSDOS	4.2K RAM
DISK BASIC	5.8K RAM
files	A 256 byte buffer must be reserved for each file, and some extra overhead space for a total of about 280 bytes per file.

DISK BASIC

Radio Shack Disk Basic adds several non disk related statements to the interpreter. The additional LEVEL II statements and functions provided by the mini disk are:

MID$ (left side of equation)

INSTR

TIME$

USR (USR0-USR9)

DEFUSR

Hexadecimal and Octal Constants

LINE INPUT

DEF FN

CMD"D"

CMD"T"

CMD"R"

MID$

MID$ can be used on the left side of an equation to replace a substring in an indicated string. The general format is similar to the format used for MID$ on the right of an equal (or relational) sign.

MID$ (string #1, I,J) = string #2

This will replace a portion of the "string" beginning at position I for J characters with the string indicated by "string #2". J is optional value. If J is not expressed, the string will replace the portion of "string #1" beginning at I for the entire length of string #2 or to the end of string #1 whichever is smaller. This limitation precludes any changes to the length of string #1.

Example:

```
10 INPUT "STRING 1 -- STRING #2 -- START -- LENGTH";B$, L$, S,K
20 MID$(B$,S,K) = L$
30 PRINT B$
40 GOTO 10
   RUN

String 1 -- String #2 -- Start -- Length?  ABCD, XY, 2,2
    AXYD
String 1 -- String #2 -- Start -- Length?  1901 DAKAR ST. W., RD, 12,
    1901 DAKAR RD. W.
String 1 -- String #2 -- Start -- Length?  GO BRONCOS GO, COWBOYS, 4,
    GO COWBOYS GO
```

INSTR - This is a string function which searches for the occurrence of one string within another string. INSTR will return the starting position of the occurrence. (This function replaces the INSTRing subroutine given in the LEVEL II manual.) The general format of INSTR is:

```
INSTR (I, string #1, string #2)
```

This will search for the first occurrence of string #2 in string #1 and if a match is found, the value returned will equal the starting position of the match.

I is an optional parameter which specifies the position in string #1 where the search is to begin. If I is greater than the length of string #1, or string #1 is null, or no match is found, INSTR returns a 0. If string #2 is null, INSTR returns I (if specified) or 1.

TIME$ - This is a 17 character string containing the date and time. The date and time are initialized by the DATE and TIME utilities of the DISK OPERATING SYSTEM. (see DOS UTILITIES chapter) The format of TIME$ is "MM/DD/YY HH:MM:SS". To print the time on the line printer type:

```
LPRINT RIGHT$(TIME$,8)
```

To print the date in the center of the screen type :

```
PRINT @540, LEFT$(TIME$,8)
```

NOTE: When the real time clock is stopped for tape loading (see CMD"T") TIME$ will not be updated by DOS. CMD"R" restarts the clock.

DEF FN - User defined functions

DEF FN designates a variable as a function

```
DEF FN variable name (variable list) = expression
```

The variable name will be the name of the function. The name can consist of any number of characters, however, the first character must

be alphabetic and only the first two characters will be recognized (as in variables). The "variable list" consists of the variables that are to be used in the function. These may be any number of legal variables, separated by a comma, depending on the number of arguments needed by the expression. The expression is the function itself. These may only be one logical line or statement in length. (Statements separated by colons are not allowed.)

For example:

```
10 DEF FNMLT(X,Y)=X*Y
20 INPUT A,B
30 C=FNMLT(A,B)
40 PRINT C
```

The function MLT multiplies two arguments. Line 30 takes the values of A and B, passes them to the user defined function in line 10 which multiplies them, and stores the result in C.

Strings may also be manipulated by functions. For example:

```
10 DEF FNADD$(A$,B$) = A$+" "+B$
20 INPUT"ENTER FIRST NAME";X$
30 INPUT"ENTER LAST NAME";Y$
40 Z$=FNADD$(X$,Y$)
50 PRINT Z$
```

In this example a dollar sign ($) was added to the function name indicating a string function. This is necessary because, just like variables, functions must indicate which variable type is to be returned - single(!) or double precision (#),integer (%), or string ($). The default is single precision.

The usefulness of the DEF FN statement becomes apparent when a particular function is used several times in a program. This will save time and memory space when performing repetitive operations.

HEXADECIMAL and OCTAL CONSTANTS - In some functions it is more convenient to use HEX (base 16) or OCTAL (base 8) constants rather than decimal numbers. These number bases are specified by the following symbol(s):

Octal - & (octal constant)

HEX - &H (hexadecimal constant)

For example:

POKE &H42E9, &HFF

would POKE the hexadecimal constant FF (255 decimal) at location 42E9 HEX (17129 decimal). Hex and Octal constants may not be used in response to INPUT statements or in DATA statements.

USR - The USR function has been expanded to allow up to 10 machine language user routines. The routines can be assembled using the TRS-80 Editor/Assembler program and loaded under the SYSTEM command or the object code can be loaded from the keyboard using the POKE statement. DEFUSR has been provided to assign the starting addresses for the routines rather than POKEing the address in a user location as described in the LEVEL II manual.

The general format for calling a USR routine from a BASIC program is:

USRn(arg)

n is an integer value from 0 to 9 and represents the routines assigned with the DEFUSR statement (see DEFUSR). This number will call one of the ten possible user routines. For example,

X=USR3(0)

will call user program #3. If a value is to be returned directly by the routine, X will contain the value produced.

As in normal Level II operations, user routines in Disk Basic are protected using the MEMORY SIZE option given at power up.

DEFUSR - Is provided to assign entry points to USR routines. This statement replaces the user location - POKE - method described in the Level II manual. The general format of the function is:

DEFUSRn = addr

n may be any number from 0 to 9 representing the 10 possible USR routines. Addr is an integer value indicating the starting address of the USR routine. 7. For example:

DEFUSR7 = &H70E9

would assign USR7 a starting address of 70E9 HEX.

For example: This program will print the numbers from 1 to 100 and then call a machine language subroutine (line 100) to "White out" the screen. The machine language routine is POKEd into memory from the data statements

```
10 DEFUSR1 = &H7D00
20 FORX=32000 TO 32013
30 READ A
40 POKE X,A
50 NEXT X
60 CLS
70 FOR X=1 TO 100
80 PRINT X;
90 NEXT X
100 X=USR1(0)
110 FOR X=1 TO 1000
120 NEXT X
130 GOTO 60
140 DATA 33,0,60,54,255,17,1,60,1,255,03,237,176,201
```

PASSING ARGUMENTS TO AND FROM USR ROUTINES - There are 2 ways to communicate from BASIC to machine language subroutines.

1. POKE the arguments into fixed locations in RAM and, after the subroutine does its work, PEEK the results back into BASIC.
2. Pass the argument as part of the USR function while using routines in BASIC to do the number conversions.

This example will give subroutine the value stored in X and return a value in Y.

BASIC

```
10 DEFUSR5= H7D00        Define entry point to user subroutine
20 INPUT X               Number 5 as 7D00 (hex)
30 Y = USR5(X)           At this point control is passed to the
                         subroutine
```

SUBROUTINE

```
CALL   0A7FH             this is a routine in ROM which will take
                         the value of X (the argument in line 30)
  main body of           convert it to an integer and store it in HL
  subroutine

JP     0A9AH             this is a routine in ROM which will take
                         the value in HL and pass it to the calling
                         Basic program as the new value of USR.  In
                         the above example Y = value of HL
```

CMD"T" - Time Out - This command must be used, either in the command mode or within a program, before any tape operations. CMD"T" turns off the REAL TIME CLOCK so timing sensitive tape commands will not be interrupted. Commands affected on CLOAD, CSAVE, INPUT#-1(-2),SYSTEM (filename)

CMD"R" - Restart Clock - Use this command to restart the clock after tape operations.

Example:

```
10 CMD"T"                turns off clock
20 INPUT#-1, A,B,C
30 CMD"R"                turns on clock
```

CMD"D" - This command loads the DOS debugger. See the DOS UTILITIES section, DEBUG command.

CLOAD? - This verify routine is not available under DISK BASIC for comparing programs CSAVEd under LEVEL II. A "BAD" message will always result. Programs CSAVEd under DISK BASIC will verify correctly.

Line Input - Causes all characters typed in to be assigned to a string variable. This is used when commas, quote marks or other delimiter may be input, as in names, addresses, etc.

Line Input "prompt string"; single variable

eg. 10 LINE INPUT "ENTER YOUR NAME";N$

would cause all characters typed in up to an (ENTER) to be assigned to N$ even though the name may contain a comma. A question mark is not printed unless it is part of the prompt string.

CLOAD may not be used with a file name under Disk Basic. For example CLOAD"A" may cause the computer to hang up. To load from tape, use the following sequence:

```
CMD"T"                  turn off clock
CLOAD
CMD"R"                  restart clock
```

CSAVE still requires a file name

RADIO SHACK LEVEL II DISK COMMANDS

The commands used with the disk system facilitate the control of disk operations. They initiate and terminate disk operations and specify the types of files to be used. They also control files in much the same way Level II Basic commands control programs.

LEVEL II Disk commands are:

OPEN

CLOSE

SAVE

LOAD

MERGE

KILL

CMD"S"

RUN"filename"

FILE NAMES

Throughout this manual references are made to filenames. These names can always consist of up to 4 descriptors, the name itself, a file extension, a password, and a drive number; in that order. All but the name itself are optional.

The name can be from 1 to 8 alpha-numeric characters with the first character being a letter. Example:

MASTERIN

PAYEMP25

DOLLAR

Z

XYZ123

The file extension can be from 1 to 3 alpha-numeric characters and is used to identify a file type. File extensions must be preceded by a slash (/). If it is not specified, the operating system uses blanks for the file extension.

Example:

/CMD	could indicate a command file
/OBJ	could indicate an object file
/SYS	system file
/DAT	data file
/BAS	basic program file

If a file extension is non-blank, it must be specified for all disk operations involving that file.

The file extension is listed beside the file name by the "DIR" directory command.

The password is a file protection feature. It can be from 1 to 8 alpha-numeric characters. If a password is assigned when a file is

created by an OPEN statement then that password must always be provided for subsequent disk operations. Without the proper password a file may not be read, copied, opened, or deleted. BE CAREFUL!!! If you forget the password, you will never be able to use that file again. All passwords are preceded by a period. Examples:

.CRIMSON

.SKY

.XMASTREE

The drive number is a digit from 0 to 3 preceded by a colon. It refers to a particular disk drive (drive 0 is the disk closest to the expansion interface, drive 3 the furthest). Unless a drive number is specified for new files, that file will be written on the lowest numbered drive in the system with available space and not write protected. If no drive is specified for input files, the operating system will search each drive for the requested file, the only loss is access speed. Examples:

:3 drive 3, the last on the cable

:2 drive 2, the second to last one

These are examples of file names using various combinations of descriptors.

MATHTEST/BAS.TEACHER:2

the file name is MATHTEST and is a BASIC program file

the password is TEACHER and is located on drive 2

INVT:1

the file name is INVT and it is on drive 1, no extension

is given and no password is required

PAYROLL/DAT.IRS

the file name is PAYROLL and it is a DATA file, the password

is IRS and you don't know which drive it is on. The drives

are searched in numerical order until one is found which has

space and is not write protected. That drive will be used.

You may have two files with the same name as long as some part of the file name (extension, or drive number) distinguishes it. For example, all the following files are uniquely different.

FILE:1	FILE/BAS:1
FILE/BAS:0	FILE/SYS
FILE:0	FILE/OBJ

The password will not make a filename unique. If a program is saved a FILE/BAS and then another attempt is made at saving the same program as FILE/BAS.PASS an FILE ACCESS DENIED error will occur. The password does not uniquely differentiate a file. A password only places a protection on a file when it is saved.

The OPEN command must be used before any disk reads or writes for data files; this data does not apply to program saves and loads. The format for using the OPEN command is as follows:

OPEN "mode", #filenumber, "filename"

The mode specifies the type of file. The following modes are used:

MODE	DESCRIPTION
O	Output mode for sequential files - write to disk
I	Input mode for sequential files - read from disk
R	Input and/or Output for Random files

Files numbers are used to identify a file that is opened. In later Input/Output operations, this number is used to identify the file and options rather than the name, mode, disk #, etc. Filenumbers are integers and range from 1 to 16. This means that 16 files may be opened at one time and used in different operations by using the file number given in the OPEN statement.

Filenumber correspond to the number of files requested with HOW MANY FILES? If you requested 6 files then your filenumber may only be the numbers 1 through 6. Likewise, the default of 3 files only allows you the filenumbers 1, 2, or 3. The # sign above is optional in the OPEN statement. No two OPENed files may have the same file number, but filenumbers may be changed by CLOSEing the file and reopening with a different file number.

The filename is an alphanumeric string which was used to identify the file when it was stored on the disk (or to name a file that you will be creating).

Example of OPEN statements:

```
OPEN "O",1, "OUTPUT"
OPEN "I",2,"INPUT"
OPEN"R",3,"FILE:1"
OPEN D$, X, N$
```

In the fourth example. variables are used to form the OPEN statement.

It is standard to insert OPEN into programs by giving it a line number. A file should only be opened with one mode at a time. Attempts to open a file with a filenumber already in use will cause a FILE ALREADY OPEN error.

CLOSE - CLOSE will terminate access to a specified file by closing out INPUT/OUTPUT to the file. The format is:

```
CLOSE filenumber,...,filenumber
```

Examples: CLOSE 1, 2, 6

Closes file indicated by 1, 2, and 6

The file numbers are optional. If CLOSE is used without filenumbers, all open files will be closed.

A CLOSE issued to a sequential output file will write the final buffer and CLOSE the file.

A NEW command will automatically close all open files as well as deleting the program and variables resident in the computer. All files must be closed before changing diskettes on a particular drive.

KILL - KILL deletes a disk file. This will release the space used by the old file for a new one.

```
KILL"filename"
```

Example: KILL"DATAFILE"

A file must be CLOSEd before you attempt to KILL it. A file that is OPEN (see OPEN) cannot be deleted (a "FILE ALREADY OPEN" error will occur if

this is attempted).

MERGE - The MERGE command combines a resident program with one located on disk. The incoming program overlays the resident program and replaces any lines having the same line number. This edits the resident program by replacing lines with the program coming in from the disk. The general format is:

MERGE"filename"

Example:

MERGE"ADDPROG"

The file must have been saved using the "A" (ASCII Format) option described under SAVE.

SAVE - Programs may be saved on disk using the SAVE command. This operation is performed using the following statement:

SAVE"filename"

Example:

SAVE"PRGRM10"

This will store the program and delete any programs of the same name. Therefore, if you edit or update a program, the current program will replace the old one.

Programs are normally stored using a compressed format. This is done automatically by the computer. Programs can be stored using an ASCII format by specifying the "A" option. This is used when program text is to be read in by another program (either MERGE or LINE INPUT). An example of such a program might be a routine to renumber lines in another program. To specify the ASCII option, use the following format.

SAVE"filename",A

Example:

```
SAVE"PRGRM20,A
```

Programs saved in ASCII will transfer more slowly than files stored in the normal binary format.

LOAD - This command loads a program stored on a disk into memory. The LOAD command uses the following form:

```
LOAD"filename",R(optional)
```

Example:

```
LOAD"PRGRM30",R
```

This loads the program specified from the disk drive specified and executes the "R" option.

The "R" option specifies the program is to be loaded and RUN. When a LOAD is issued without an R option all files are automatically closed and all resident memory is deleted (like NEW). If the R option is used all data files are kept onpen and only program lines and variables are deleted from memory.

```
RUN"FILENAME"
```

The command performs the same function as the LOAD command with an R option specified. The following command are equivalent:

```
LOAD"PROGRAM",R
RUN"PROGRAM"
```

CMD"S" - This command will return control to the disk operating system. When this command is used in BASIC the computer will respond:

```
DOS READY
```

SEQUENTIAL DISK DATA FILES

Sequential input and output is the simplest form of disk data storage. The statements are very similar to one used for tape storage. The statements used in sequential I/O operation are:

PRINT#	LINE INPUT#
PRINT USING	
INPUT#	EOF

PRINT# - Is used to write data to a sequential output file. The format is very similar to the statement used in the PRINT statement for creating cassette tape files.

PRINT# file number, variable or exp.;...; variable or exp.
This will write the data specified in the variable/expression list to the file indicated by the file number. The file number is the one specified in the OPEN statement for that file.

Example: PRINT#1,A;A$;Z;A$;LEFT$(Z$,5)
This will write the specified information to the file indicated by filenumber 1. ASCII characters can be inserted using the CHR$ statement and ASCII codes. This method is used when a string contains characters or control characters other than alphanumerics such as commas, linefeeds, etc.

eg/ PRINT#1, CHR$(34);X$;CHR$(34)....
If the string is alphanumeric only -- commas can be inserted

PRINT#1, A$;",";B$;",";...

PRINT USING - Allows you to write to a sequential file using a specified format. The formats are indicated by the format specifiers used in the BASIC PRINT USING statement. The format is:

PRINT#filenumber,USING"format";Variable/expression list

Example: PRINT#1,USING"**$###.##";121.129

will print *$121.13 to the disk file.

INPUT# - Used to read back sequential data written by the PRINT# instruction.

INPUT# filenumber, variable, variable ....

This will read data from the disk into the specific variables. The file number is specified at OPEN time.

Example:

```
INPUT#3,A,B,C
```

This will read data from the disk assigning A to the first value read, B to the next etc.

There are some important differences between disk and tape files. When PRINTing strings, it is necessary to separate them with explicit commas as below in line 30.

```
10 OPEN"O",1,"NAMES"
20 A$="JOHN":B$="SMITH
30 PRINT#1,A$;",";B$
40 CLOSE:OPEN"I",1,"NAMES"
50 INPUT#1,A$:PRINTA$
60 CLOSE:END
```

When this program is run it will print JOHN. If line 30 were PRINT#1,A$;B$ then this program would print JOHNSMITH. The commas are needed between strings so Disk Basic will know where a string ends and begin. If you forget the commas you could easily get a READ PAST END FILE error. Note the commas are not needed between numeric values. PRINT#1,A;DC;B is a valid statement.

Another difference is that PRINT# statements need not correspond exactly to the INPUT# statements. With tape operations, a PRINT#-1,A,B,C requires an INPUT#-1,A,B,C to read the three values. This is not so with disk. Observe the following program:

```
10 OPEN"O",1,"DATA"
20 A=1:B=2:C=3:D=4
30 PRINT#1,A,B,C,D:CLOSE
40 OPEN"I",1,"DATA"
50 INPUT#1,A,B,:PRINTA,B,
60 INPUT#1,A,B:PRINTA,B:CLOSE
> RUN
 1            2            3         4
```

NOTE that the PRINT on line 30 does not match the two INPUTs on lines 50 and 60. Yet the program works correctly.

In order to simulate a RESTORE command for a disk file it is necessary to CLOSE the file and then reOPEN it again. This sets all reading (INPUT#) back to the beginning of the file.

LINE INPUT - Will input any string from disk or the keyboard, ignoring commas or quotes until a carriage-return or 255 characters are read. From the terminal, Input will be requested without a question mark being typed. You may type quotes ("), commas (,), or linefeeds (↓) until an ENTER is typed.

```
10 CLEAR 300
20 LINEINPUT A$
30 PRINT A$
```

Try the above program with linefeeds and commas. Note that a BREAK will stop the program during a LINE INPUT.

The following program will read itself off of disk and list itself on the monitor screen.

```
10 CLEAR 500
20 OPEN"I",1,"PROG"
30 FORI=1TO5
40 LINEINPUT#1,A$
50 PRINT A$:NEXT
> SAVE"PROG",A
> RUN
```

Be sure to SAVE the program as an ASCII file before running it. The formats for LINE INPUT are:

LINE INPUT string variable (keyboard)

and LINE INPUT#filenumber, string variable

EOF - Denotes the end of a file. When inputing data from a disk file, this will detect the end of the data file. The EOF works as a logical function to test for the last record of the file. It can be used in two ways:

Variable = EOF(file number)

eg/ X=EOF(1)

assigns X a -1 if no more data, 0 if more data in file. It can be also used with an IF statement.

IF EOF(1) GOTO 999

If end of file then branch to 999, if not continue on the next line. Example:

```
10 OPEN"I",1,"FILE1"
20 FOR X=1 TO 20
30 IF EOF(1)THEN 60
40 INPUT #1,A(X)
50 NEXT
60 PRINTX;"RECORDS READ"
70 CLOSE
```

Files are automatically allocated 1½K If more space is used, another 1½K is allocated etc. Therefore, the minimum space allocated for a file is 1280 bytes and files always acquire memory1½K at a time instead of smaller units (bytes at a time).

Random files offer two distinct advantages over sequential files. Sequential accesses require record by record data search, where Random accesses provide immediate retrieval of desired data. Sequential data is stored ASCII format while Random data is stored in a compressed binary format. Sequential files may only be opened in an input or output mode, while a Random file may be written to or read from at will. Data can easily be read, modified, and written back to disk, changing old information.

A 256 byte record is the primary element in a random file. A file can contain as many as 329 of these records. A record number between 1 and 329 is associated with each record. This serves as the record identifier and all attempts to read or write records include this identifier. Of course, the first record in the file has a record number of 1, the second of 2, and so on. If you wish to access any record in a file, all that is needed is its record number for immediate access to that data.

Random records are read and created in a BUFFER. A buffer is a 256 byte area of memory where records are built one variable at a time. The buffer is then written onto disk. The reverse is true for reading random records. The buffer is filled from disk then variables are pulled from the buffer one at a time.

This breaking up of the buffer into variables and pulling out data is called "fielding". An example will clarify this. Suppose a file is to contain a mailing list of names and addresses. Record one contains the following:

ATKINSON MIKE 1000 JACKRABBIT SCOTTSDALE AZ

If this information were read from disk into the buffer, how do we know where the name starts, and the address begins? The answer is that you MUST know and MUST inform the computer. Using a FIELD statement

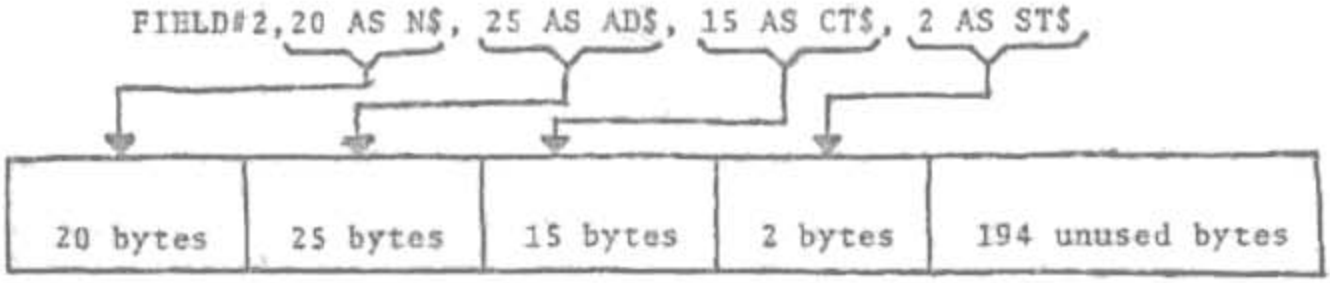
the computer is informed of the format of the data.

```
OPEN"R",2,"MAIL"
FIELD#2,20AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
```

This FIELD statement specifies the format for file #2 as having the first 20 characters referred to AS N$(name), the next 25 characters referred to AS AD$(address), the next 15 characters referred to AS CT$ (the city) and the next 2 characters AS ST$(state). In general the FIELD statement specifies the length of each item within a record and which string variable they are assigned. Be careful with string variables used in a FIELD statement. As you will see, they must be treated differently than usual string variables.

If the lengths of each item are added together the total length of the data is 20+25+15+2=62 characters. But if the buffer is 256 characters long, what are the other 194 characters doing? They are wasted! Each time a name and address are PUT onto disk 194 bytes are unused. Thus three-fourths of the files are wasted. Later a method will be shown to store four addresses in the buffer at once. This would waste only 8 characters out of the 256 byte buffer.

The following is a picture of the buffer and a FIELD statement and how they interact.



Buffer for file #2 (256 bytes total)

All variables in the buffer are stored as strings. There are special commands in DISK BASIC to convert integers, single precision, and double precision numbers to strings and back to numbers again.

Integers in the range of -32768 to +32767 are stored as two byte binary numbers. Therefore, when converted to a string, these will be stored as two byte strings in the random data file.

Single precision numbers are stored as 4-byte binary numbers. They are converted to 4 byte strings when stored on the disk.

Double precision numbers require 8 bytes and therefore require 8-byte strings to store them.

This will become important when you are dividing the buffer into fields. Integer values will be fielded 2 bytes, single precision 4 bytes and double precision will be assigned 8 bytes for storage.

CREATING RANDOM FILES ON DISK

Files are created by placing the appropriate data in the buffer and then storing the contents of the buffer on the disk as a record. Since each record is assigned a number, any record can be accessed individually by referring to its record number.

The first step is to OPEN the file under the R(Random access) option.

OPEN "R", filenumber, "filename"

As in sequential access, the filenumber is used to refer to a particular data file. This will also assign a buffer to the file for use in accessing the disk file from your program.

FIELD - The next step is to divide the random buffer into segments, which will hold the particular elements of data. Each FIELD statement refers to a "type" of record. If all the records on a disk will look alike, only one FIELD statement will have to be used for all the records. The best way to determine the fields for a buffer is to write down a representative record and determine the maximum length for each element. Suppose we wished to store information in a file which contained a NAME, ADDRESS, PHONE NUMBER, ID NUMBER and a DOLLAR AMOUNT. A representative record might look like this:

DATA	NUMBER OF BYTES	MAXIMUM NUMBER OF BYTES	TYPES OF DATA
JOHN DOE	8	20	string
999 W. 8TH AVE	15	20	string
999-9999	8	8	string
1056	2	2	integer
132.84	4	4	single precision

Since most names are longer than 8 characters, the field should be long enough to contain most long names (20 characters is standard). The

general format for the field statement is:

```
FIELD# filenumber, field size AS buffer string variable,...,
    field size AS buffer string variable
```

A pound sign (#) is optional before the filenumber.

The FIELD statement for the example would be:

```
FIELD 1, 20 AS NAM$, 20 AS ADR$, 8 AS PH$, 2 AS NI$, 4 AS NS$
```

More than one type of FIELDed data can be used in the same data file. This is done by refielding the buffer by executing another FIELD statement. If several different fields are used in the same file or program, it is convenient to put the field statements in subroutines and execute them when a particular type of data is being sent. This will be shown shortly.

Don't let the FIELD statement trick you! The buffer is 256 bytes long but remember that BASIC only allows a string to be 255 bytes. Thus the following is illegal

```
FIELD 1, 256 AS Q$
```

or

```
FIELD 2, 128 AS Q$, 128 AS R$
```

Note that no individual field may exceed 255 bytes and that the sum of the lengths of all elements may not exceed 255.

Moving Data to the Buffer - After the buffer has been FIELDed, you then place the appropriate data in the fields. Several statements facilitate and control this transfer.

STRINGS - LSET and RSET -- Strings are placed into the buffer using two statements, depending on whether you want the string right or left justified. A left justified string is one that starts from the left and fills the field to the right. If the string is longer than the field, the rightmost

position of the string will be lost. If the string is smaller than the allotted field, it is "padded" with blanks on the right. A right justified string is just the opposite. The field is filled from right to left and padded with blanks on the left if smaller and cut off on the left if larger. These assignments are made using LSET and RSET.

LSET buffer string variable = string expression

RSET buffer string variable = string expression

For example, in the Field statement, NAM$ was fielded to 20 bytes. The field could be assigned a string using either LSET or RSET in the following manner.

```
A$="DOE":B$="JOHN"
LSET NAM$ = A$+", "+B$
```

The buffer field would look like this (Ø represents a blank):

DOE,ØJOHNØØØØØØØØØØØØØØ

```
RSET NAM$ =A$ +", "+B$
```

would result in: ØØØØØØØØØØØØØØDOE,ØJOHN

NUMERIC VALUES - MKI$, MKS$ and MKD$ -- Since numbers are stored in a RANDOM file as strings, 3 statements are provided to make the conversions and send them to their respective fields in the buffer.

MKI$ - This function converts an integer to a two byte string and sends it to its assigned field in the random buffer. The general format is:

MKI$ (integer or integer variable)

For example, assume the variable NI$ has been fielded for a two byte integer string: to convert the integer to a two byte string and place it in the field we could write:

```
A% = 9999
LSET NI$ = MKI$(A%)
```

This would convert the contents of the integer variable A% to a two byte string and field it in NI$. The integer value must be in the range of -32768 to +32767. Either LSET or RSET must ALWAYS be used when moving anything to the buffer.

MKS$ - This function converts single precision values to a 4 byte string. and places it in the indicated field in the buffer. The general form of the statement is:

MKS$(single precision variable or value)

If the 4 byte FIELD was defined by NS$ the statement might look like this:

```
A! = 9999.99
RSET NS$ = MKS$(A!)
```

This would convert the single precision number indicated by A! to a 4 byte string and place it in the field specified by NS$.

MKD$ - This function converts a double precision value to an 8 byte string and places it in a specified field. The format is:

MKD$(double precision variable or value)

Example: if the 8 byte field was indicated by ND$

```
A#=3.14159262SD0
LSET ND$ = MKD(A#)
```

would convert the double precision value indicated to an 8 byte string and place it in the buffer field indicated by ND$.

PUT - The statements we have described so far, set up the buffer and place information in it. When the buffer contains the information you wish to store as record on the disk, a single statement is executed to transfer the contents of the buffer to the disk file. The PUT statement assigns the data in the buffer a record number and stores it in the specified file. The format is:

PUT filenumber, record-number

The filenumber indicates the file which was opened to contain the data and the record number is the number assigned to one of 329 records. The record number is optional. If the record number is not specified, the data in the buffer will be initially written to record 1 and thereafter the records will be written into the next record number in increments of one when each PUT is executed.

Example:

PUT 1

When a record is PUT, the disk space for all record numbers from 1 up to the record number use is reserved. Thus a PUT 1, 350 will cause a DISK FULL error even though only one record of meaningful data was written. This is because space for records 1 through 349 was set aside.

LOF - This function will return a value indicating the last record number in a given file. It is especially useful when adding records to a file that was previously constructed. The general format of the LOC statement is:

LOF (filenumber)

Example:

PRINT LOF(1)

will return a value for the last record in file 1. This prevents reading past the end of the file and reading meaningless characters into the buffer.

RETRIEVING DATA FROM A RANDOM FILE

Getting data back from a random data file is very similar to the methods used for storing it. However, the process is reversed.

FIELD - This statement is used to prepare the buffer for data coming in from the data file. (The format for the FIELD statement is identical for the field used for storing the data)

FIELD# filenumber, field size AS buffer string variable,...,
field size AS buffer string variable

The same field statement can be used to retrieve the data from disk as the one used to store it there (or one identical to it) if the data is to be read in the same manner it was written. However you may read data in a different format than the one used to store it.

For example, if the first 40 characters of a record were written as 3 separate strings, they could be read back as one string:

FIELD 1, 40 AS THE$,...etc

In some operations, only one piece of data from a record may be required. If the required information is at the beginning of the record, it is rather simple to field only the required number of characters as one variable and the remainder as another, dummy variable.

However, if the required information is located in the middle of the record, several methods can be used to fetch the data. One method is to assign the unneeded data preceeding the data to a dummy variable. in a field statement then field the information you want and field the remaining data into another variable. If you had a field which looked like:

N$	SS$	ID$	PH$
A NAME	A SOCIAL SECURITY NUMBER	I.D. NUMBER	PHONE #

and you needed only the I.D. NUMBER(ID$) you could do this by fielding N$ and SS$ as one string, then fielding ID$, then PH$. ID$ would then be fielded as a separate number and could be read individually.

Suppose N$(name) and SS$(social security number) total to 25 characters. In order to get to the ID number in the 5th record, the following program can be used:

```
10 CLEAR 500:REM NEED EXTRA STRING SPACE
20 OPEN"R",2,"DATA"
30 FIELD 2,25 AS DUMMY$, 6 AS ID$
40 GET 2,5
50 PRINT ID$
```

This will print out the 6 character ID number from the 5th record. The first 25 characters are assigned to DUMMY$ and not used. But this moves through the buffer so that characters 26 through 31 can be used.

Another method would be to use the FIELD statement used when storing the data and then inputing only the string containing the data you need and ignore the rest.

GET - This statement is used to copy a record from the disk into the fielded random buffer. The format is:

```
GET  filenumber, record#.
```

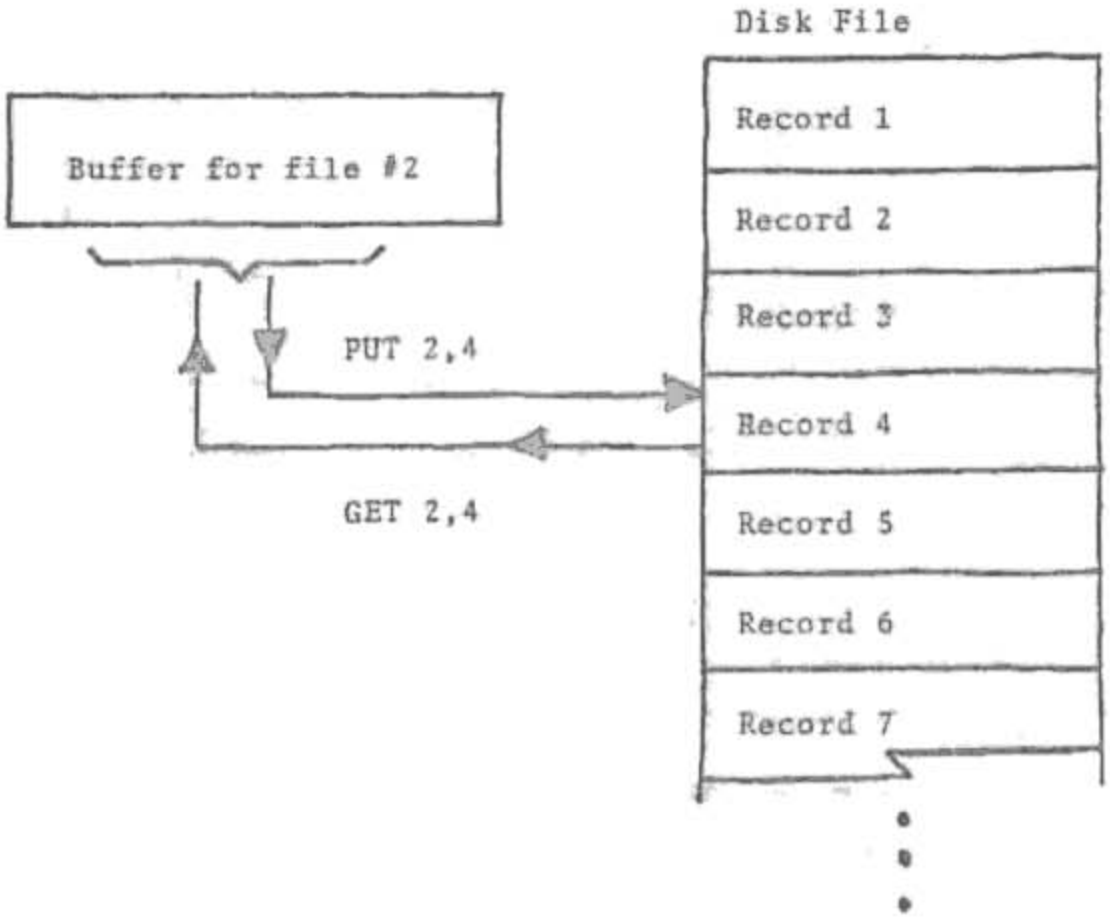
Example:

```
GET 1, 100 or GET X, R
```

This will copy the record from the appropriate file to the file buffer.

The record number is optional. If no record numbers are used, the first record accessed with a GET will be record 1. The next record will be 2 and so forth sequentially until the end of the file.

PUT/GET DIAGRAM



RETRIEVING DATA FROM THE BUFFER

STRINGS - Strings can be read from the buffer simply by referencing the variable of the appropriate field. For example:

If the string AA$ was fielded into the buffer from the disk data file using:

```
FIELD 1, 10 AS AA$
```

the string could be fetched from the buffer by using:

```
B$ = AA$
```

This would assign the contents of the buffer field AA$ to the string variable B$. Since the original string was stored as either a right justified (RSET) or left justified (LSET) string, B$ and AA$ will also have the same configuration.

NUMERIC VALUES - Since numeric values are stroed as 2, 4 or 8 byte strings (using MKI$, MKS$, MKD$); they must be reconverted to numbers when they are fetched from the buffer. Again, there are three statements which will handle the 3 types of numbers: integers, single precision and double precision.

CVI - This statement will convert two byte strings representing integers into a base 10 integer value and remove it from the buffer. The general format is:

```
CVI(2 byte string)
```

Example:

```
A% = CVI(NI$)
```

This will convert the string represented in the field by NI$ to its integer equivalent and store it under the integer variable A%.

An attempt to convert a string smaller than 2 bytes will result in a FC error. If the string is more than two bytes long, the extra bytes

will be ignored resulting in a unpredictable value in some cases.

CVS - This function will convert a 4 byte string into its equivalent single precision value and retrieve it from the buffer. The format is as follows:

    CVS(4 byte string)

Example:

    A! = CVS(NS$)

This will convert the 4 byte string represented in the buffer by NS$ to a single precision number and store it under the single precision variable A!.

If the string is less than 4 bytes an FC error will occur. If the string is greater than 4 bytes, the extra bytes (to the right) will be ignored which may return an incorrect value.

CVD - This function converts an 8 byte string into a double precision value and retrieves it from the buffer. The format is:

    CVD(8 byte string)

Example:

    A# = CVD(ND$)

This example will convert the 8 byte string represented in the buffer by ND$ into its double precision value and store it under the double precision variable A#.

SUB-RECORD USE

Now what does one do about wasted disk space? Referring back to the mailing list problem there were 194 bytes unused. Since only 62 bytes are needed, there is room for four(4) records within the buffer. Four(4) times sixty-two(62) gives 248 characters. Thus only 8 bytes are unused per record.

Each physical record (a buffer full of data) contains 4 sub-records numbered 0 through 3. At the beginning of physical record 1, the 1st address is found. At the beginning of the second physical record is the 5th address. The 5th address is termed as being the 5th "logical record". Given the logical record number (ie, which piece of data), the physical record and sub-record numbers can be calculated.

```
LR = the logical record number
physical record = INT((LR-1)/4)+1
sub-record      = LR-4*((LR-1)/4)-1
```

In order to FIELD this, it appears that four separate FIELD statements are needed. These would be:

```
FIELD 1, 0*62 AS D$, 20 AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
FIELD 1, 1*62 AS D2, 20 AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
FIELD 1, 2*62 AS D$, 20 AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
FIELD 1, 3*63 AS DS, 20 AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
```

These represent the FIELD statements for the subrecords 0 through 3 respectively. Note the use of the dummy variable D$, which is assigned the leading information that is not wanted.

Observing the pattern to these FIELD statements, a generalized FIELD can be written if the sub-record number (SR) is known.

```
FIELD 1, SR*62 AS D$, 20 AS N$, 25 AS AD$, 15 AS CT$, 2 AS ST$
```

A full working program for manipulating the mailing list file MAIL is listed below.

```
100 CLEAR 1000:CLS:CLOSE
110 PRINT:INPUT"TYPE 1 TO WRITE, 2 TO READ ";N
120 OPEN"R",1,"MAIL"
130 CLS:ON N GOTO 200,300
200 PRINT:INPUT"ENTER LOGICAL RECORD NUMBER";LR
210 IF LR=0 THEN 100
220 GOSUB 500:PR=INT((LR-1)/4)+1
230 GET 1,PR:PRINT"PHYSICAL RECORD # =";PR:PRINT
240 PRINT"NAME";TAB(20);:INPUTA$:LSET N$=A$
250 PRINT"ADDRESS";TAB(20);:INPUTA$:LSET AD$=A$
260 PRINT"CITY";TAB(20);:INPUTA$:LSET CT$=A$
270 PRINT"STATE";TAB(20);:INPUTA$:LSET ST$=A$
280 PUT 1,PR:GOTO 200
300 PRINT:INPUT"ENTER LOGICAL RECORD NUMBER";LR
310 IF LR=0 THEN 100
320 GOSUB500:PR=INT((LR-1)/4)+1
330 GET 1,PR:PRINT"PHYSICAL RECORD # =";PR:PRINT
340 PRINT"NAME";TAB(20);N$
350 PRINT"ADDRESS";TAB(20);AD$
360 PRINT"CITY";TAB(20);CT$
370 PRINT"STATE";TAB(20);ST$:GOTO 300
490 REM SUBROUTINE TO FIELD PROPER SUB-RECORD
495 REM      WITHIN THE PHYSICAL 256 BYTE RECORD
500 SR=LR-4*INT((LR-1)/4)-1
510 PRINT"SUB-RECORD # =";SR
520 FIELD 1,SR*62 AS D$,20 AS N$,25 AS AD$,15 AS CT$,2 AS ST$
530 RETURN
```

Note that the FIELD statement has been placed in a subroutine (lines 500-530). Given the logical record number (LR), the program calculates which physical record (PR) the logical record is found in. This record is read into the buffer (GET 1, PR). Next the entire buffer is FIELDed so that N$, AD$, CT$, and ST$ refer to the desired sub-record (SR).

This program will allow creation, retrieval, or modification of any logical record. When the program is RUN, the user is asked if READ or WRITE is desired. The logical record number is then asked for and the data is displayed for READ. For WRITE the input of name, address, city, and state are requested, and the new data is written onto disk. Type a logical record number of zero to get back to the read or write question. At this point pressing BREAK will stop the program. Note that the program can change a sub-record with in a physical record without harming any of the data in the other sub-records.

Typical RUN follows:

```
>RUN

TYPE 1 TO WRITE, 2 TO READ? 1


ENTER LOGICAL RECORD NUMBER? 1
SUB-RECORD # = 0
PHYSICAL RECORD # = 1

NAME                ? JOHN DOE
ADDRESS             ? 111 ANYSTREET
CITY                ? ANYTOWN
STATE               ? TX

ENTER LOGICAL RECORD NUMBER? 6
SUB-RECORD # = 1
PHYSICAL RECORD # = 2

NAME                ? JOHN SMITH
ADDRESS             ? 1011 DISK DRIVE
CITY                ? SOMEWHERE
STATE               ? CA

ENTER LOGICAL RECORD NUMBER? 0


ENTER 1 TO WRITE, 2 TO READ ? 2


ENTER LOGICAL RECORD NUMBER? 6
SUB-RECORD # = 1
NAME                JOHN SMITH
ADDRESS             1011 DISK DRIVE
CITY                SOMEWHERE
STATE               CA

ENTER LOGICAL RECORD NUMBER? 0


TYPE 1 TO WRITE, 2 TO READ ? (BREAK)
BREAK AT 110
READY
>
```

DISK ERRORS

Disk BASIC expands LEVEL II's 2 letter error messages into full words. An M O ERROR becomes a MISSING OPERAND ERROR, etc. There is an additional set of messages which indicate errors relating to disk operations. These errors could potentially destroy data files if perpetuated, so error trapping is not supported with disk errors. All other errors may be trapped by "ON ERROR GOTO" just as in LEVEL II.

ERROR MESSAGES

CODE	ERROR MESSAGE	EXPLANATION
50	FIELD OVERFLOW	More than 255 bytes were allocated to a Random FIELD
51	INTERNAL OVERFLOW	An error has occured in the disk operating system itself or a disk I/O fault
52	BAD FILE NUMBER	A filenumber specified has not been defined or defined under a different option
54	FILE NOT FOUND	An attempt to access a file which does not exist
55	BAD FILE MODE	An attempt to perform a sequential operation or a random file or vise-versa
57	DISK I/O ERROR	An error occured in data transfer between the system and disk
58	FILE ALREADY EXISTS	An attempt to rename a file (with RENAME) when new name has been used for another file
59	SET TO NON-DISK STRING	LSET or RSET used for a string variable when not fielded.
61	DISK FULL	All available space has been utilized on a particular disk
62	INPUT PAST END	An attempt to read more data from a sequential file than exists
63	BAD RECORD NUMBER	Record number assigned to a Random record out of range (1-340)
64	BAD FILENAME	An attempt to assign an illegal filename
65	MODE MISMATCH	A Random file was opened under sequential mode or vise-versa
66	DIRECT STATEMENT IN FILE	A direct statement was read when loading a program stored in ASCII
67	TOO MANY FILES	An attempt was made to create more than 48 files on a disk

The operating system is in control when the message DOS READY is displayed on the screen. This mode is initiated automatically on power up and when CMD"S" is executed under Disk Basic. TRSDOS contains several utility programs which are available anytime DOS READY is displayed. None of these service routines are available from DISK BASIC, and no utilities work after CMD"T" has been executed.

Files are referenced with the notation filename1, filename2. A file name may be followed by the extension, drive number, and password if necessary (described in the File Names section of the manual).

AUTO filename1

Load any CMD file or any utility on power-up, (it will not load BASIC programs because at POWER UP, BASIC itself has not been loaded yet). This is an extremely valuable utility for dedicated applications of the TRS-80. Typing AUTO and the name of a program such as BASIC will cause BASIC to be loaded automatically everytime the TRS-80 is turned on. Only one parameter is allowed.

NOTE: If AUTO is told to load a flakey program, the computer will "hang-up" while trying to load the bad program, and appear to stop. In this case use the MANUAL OVERRIDE. Hold down the ENTER key while the TRS-80 is turned on, this will supress the action of the AUTO command. Then type AUTO and press ENTER. AUTO followed by no command will return the TRS-80 to the normal power-up sequence.

BACKUP

Backup copies one diskette to a blank diskette. The source drive will be requested. Reply with a number from 0 to 3 (Drive 0 is the closest drive to the expansion interface). The destination drive will also be requested. The destination drive must contain the blank diskette - formatted or unformatted, BACKUP will automatically format it. If the disk contains any data, BACKUP will be cancelled. To reuse a diskette either pass a magnet over it or preferably erase it with a bulk tape eraser, such as Radio Shack, Cat. #44-210. The creation date must then be supplied in the form MM/DD/YY. BACKUP will format (if necessary), verify, and copy.

When complete, BASKUP re-boots the DOS. If there is only one disk on your system, reply 0 (zero) to both SOURCE DRIVE and DESTINATION DRIVE requests. The system will perform a one disk backup which may require swapping diskettes back and forth several times.

COPY

COPY filename1 to filename2 creates a duplicate of filename1 under the name and descriptors specified by filename2. Example:

COPY JOURNAL3:1.XYZ TO HISTORY:3

A file named JOURNAL3 on drive 1 with password XYZ is duplicated onto drive 3 under the name of HISTORY. Copy BASIC programs by LOADing from one diskette and SAVEing to another.

DOS DEBUGGER

The debugger can be invoked under the operating system by typing DEBUG. The debugger will be executed when 1) the BREAK key is pressed 2) a program is loaded. Under BASIC, type CMD"D" and the debugger will execute immediately, type G and press ENTER to return to BASIC. CMD"D" will not work when the REAL TIME CLOCK has been turned off (CMD"T") for cassette operations.

The following commands control the operation of the debugger:

D nnnn	Display memory at address nnnn
M nnnn ↓ xx	Modify address nnnn to xx; space bar increments address
R rr ↓ nnnn	Load register pair with nnnn
X	Normal display mode (registers and memory) also terminates any incomplete entry
S	Full screen display mode (memory only)
A	Display all memory in ASCII
H	Display all memory HEX
;	Increment memory display 1 block
-	Decrement memory display 1 block
G nnnn (,bbbb,cccc)	Go to memory address nnnn, optional breakpoints bbbb and cccc. If nnnn is not specified, execution resumes at last breakpoint
I	Step one instruction at a time
C	Single call step is like I except CALLS are executed in full
U	Continuously update the display stop by holding down X or pressing BREAK key
G(ENTER)	Returns to BASIC if entered by CMD"D"

DIR↓:(DRIVE NUMBER) - displays all file names (with extentions) located on the specified drive number.

Example:

DIR↓:1 returns the names of all files on Drive 1

FORMAT - Formats a new or magnetically erased diskette to the TRSDOS standard of 35 tracks, 10 sectors at 256 bytes. Previously used diskettes must be erased with a magnet or preferably a high quality bulk type eraser such as the Radio Shack Cat. #44-210. All tracks are verified; unusable sectors are marked unavailable to the system. A drive number is requested by FORMAT, reply with a number between 0 and 3. Drive 0 is the

disk closest to the expansion interface. It requests the diskette name, reply with any name up to 8 letters. This name will be printed when directories are printed (DIR). Next input the date in the form MM/DD/YY. Finally input the master password. The password can be any word up to 8 characters. It will be used in later versions of DOS to recover files whose passwords have been forgotten. The next inquiry provides for TRACK LOCKOUT. If you don't wish to lock out any tracks, reply N. If the diskette is damaged in a known section, you can use the good part and lockout the bad by replying Y. WHICH TRACKS?, reply with individual track numbers separated by commas or a range of track numbers separated by a dash (-). FORMAT THE LOCKED OUT TRACKS? reply Y or N, and format begins.

CLOCK

displays the time on the video monitor

It can only be removed by power off or reset in CMD"T" Disk Basic

TIME␣hh:mm:ss	sets the time of day
DATE␣mm/dd/yy	sets the date
TRACE	prints the Program Counter on the video display. It can only be removed by power off or reset.

DOS ERROR MESSAGES

CRC ERROR - there is a flaw in the diskette. The track is locked out and cannot be used by the system. Affects disk capacity but not disk performance.

FILE ACCESS DENIED - The file exist but the correct password wasn't provided.

FILE NOT FOUND - the file does not exist or the name is incomplete. Remember - file extensions (/BAS/SYS etc) must be given if the file was created with an extension.